ncdjango Documentation

Release 0.4.0

Conservation Biology Institue

Oct 01, 2017

Contents

Ncdjango turns Django projects into map servers backed by NetCDF datasets. It can be added Django project to provide various web interfaces to NetCDF data and geoprocessing tools written in Python which operate on NetCDF data.

CHAPTER 1

Getting Started

Requirements

- Python 2.7, Python 3.5+
- Django 1.8 - 1.11
- `clover` 0.2.0 (https://github.com/consbio/clover)
- `numpy` (http://www.numpy.org)
- `django-tastypie` 0.13.x (https://django-tastypie.readthedocs.io)
- `djangorestframework` (http://www.django-rest-framework.org)
- `netCDF4-python` (http://unidata.github.io/netcdf4-python)
- `pyproj` (https://github.com/jswhit/pyproj)
- `fiona` (http://toblerity.org/fiona/README.html)
- `shapely` (https://pypi.python.org/pypi/Shapely)
- `ply` (https://pypi.python.org/pypi/ply)
- `celery` (http://www.celeryproject.org)
- `Pillow` (https://pypi.python.org/pypi/Pillow)
- `six`

Installation

Once the dependencies are installed, you can install ncdjango with:

```
$ pip install ncdjango
```

Note: The *clover* dependency is not the same as the *pip* package of the same name. *clover* must be installed from https://github.com/consbio/clover. The correct package should be installed by *ncdjango*, but in case of problems, it's good to check that you have the correct one.

Setup

1. Create a new Django project if you don't already have one.
2. Add `ncdjango`, `tastypie`, and `rest_framework` to your `INSTALLED_APPS` setting.
3. Modify your `settings.py` to specify the root location of your datasets:

```
NC_SERVICE_DATA_ROOT = '/var/ncdjango/services/'
```

4. Modify your `settings.py` to specify the location to store temporary files (uploads):

```
NC_TEMPORARY_FILE_LOCATION = '/tmp'
```

5. See *Settings* for additional options.
6. Add the following to your project's `urlpatterns`:

```
url(r'^', include('ncdjango.urls'))
```

Note: You can modify this URL pattern if you want all the ncdjango and web interface URLs grouped under a common path.

Publishing Services

Todo

CHAPTER 2

Interfaces

Ncdjango has two built-in interfaces. The first is a partial implementation of the *ArcGIS Server Rest API* (http://resources.arcgis.com/en/help/rest/apiref/index.html?mapserver.html). The second is a simple *data* API for querying things like value range, classifications of data, and data through time (for time-enabled datasets) at a single point.

You can also add your own interface, which is explained in *Adding your own interface*.

ArcGIS REST Interface

Todo.

ArcGIS REST Extended Interface

Todo.

Data Interface

Todo.

Adding your own interface

Todo.

CHAPTER 3

Geoprocessing

The geoprocessing module provides a framework for providing a web interface to geoprocessing jobs which operate on NetCDF data. The core components are: *tasks*, which have defined inputs and outputs and perform some function; *workflows*, which are pipelines of tasks; and a *web API* to allow clients to submit a job with inputs for processing, monitor the job status, and retrieve outputs upon completion.

Getting Started

This tutorial covers the basics of creating a task, making it available through the REST API, and running it through that API.

Creating a task

First, let's create a task:

```
from ncdjango.geoprocessing.params import IntParameter
from ncdjango.geoprocessing.workflow import Task

class SumInts(Task):
    name = 'sum_numbers'

    inputs = [
        IntParameter('int1', required=True),
        IntParameter('int2', required=True)
    ]

    outputs = [
        IntParameter('sum')
    ]

    def execute(self, int1, int2):
        return int1 + int2
```

Note: The `name` property is not strictly required except when serializing workflows to JSON or looking up tasks by name.

This task take two integers, adds them together and returns the result. When this task is called, it will automatically check for required inputs and validate the types of incoming parameters.

Running the task from Python

We can run our task from Python, by calling an instance of it:

```
>>> t = SumInts()
>>> result = t(int1=3, int2=5)
>>> result
<ncdjango.geoprocessing.params.ParameterCollection object at 0x11fa830b8>
>>> result['sum']
8
```

Note: Tasks must be called with keyword arguments. Positional arguments are not allowed.

Note: Tasks always return a `ParameterCollection` object, which can be used like a dictionary to retrieve actual result values.

We can also get our task class by name:

```
>>> Task.by_name('sum_numbers')
<class 'SumInts'>
```

This is useful when using tasks as plugins, in which case their locations may be unknown.

Registering the task with the web API

Now that we have a working task, let's make it accessible over the web. To do this, we'll need to add an entry to the project `settings.py` file:

```
NC_REGISTERED_JOBS = {
    'sum_numbers': {
        'type': 'task',
        'task': 'myapp.ncdjango_tasks.SumNumbersTask'
    }
}
```

This tells ncdjango to add our task to the web API as a job called `sum_numbers`.

Running the task from the web

Once the task is registered as a job, we can run it through the REST API. Open http://127.0.0.1:8001/geoprocessing/rest/jobs/ in your browser to interact with the API. Submitting a job requires two fields: the job name (`sum_numbers` in our case) and the job inputs, as a JSON object. For example:

```
{
    "int1": 3,
    "int3": 5
}
```

You will receive a response like this:

```
{
    "uuid": "aa346c90-68e5-4d19-a7f3-a54f6b87ec34",
    "job":"generate_scores",
    "created": "2016-09-02T23:36:10.768937Z",
    "status": "pending",
    "inputs": "{\"int1\": 3, \"int2\": 5}",
    "outputs": "{}"
}
```

Now we can use the `uuid` value to query the job stats as it runs. The status will move from `pending` (the job has been queued) to `started` (the job is running) and finally to `success` (the job is done).

```
http://127.0.0.1:8001/geoprocessing/rest/jobs/<uuid>/
```

```
{
    "uuid": "aa346c90-68e5-4d19-a7f3-a54f6b87ec34",
    "job": "generate_scores",
    "created": "2016-09-02T23:36:10.768937Z",
    "status": "success",
    "inputs": "{\"int1\": 3, \"int2\": 5}",
    "outputs":"{\"sum\": 8}"
}
```

By parsing the returned JSON object once the job has completed, we can access the output value from the task.

Note: Geoprocessing jobs will not run unless celery has been configured for the project and a celery worker is running and consuming tasks. http://docs.celeryproject.org/en/latest/django/first-steps-with-django.html

---

Tasks

Tasks represent a unit of work in the geoprocessing framework. These can be built-in, or user-defined.

Basic Task

A basic task has a name, input and output parameters, and an `execute` method.

```
class MultiplyArray(Task):
    """ Multiply the values in an array by a given factor """

    name = 'multiply_array'

    inputs = [
        NdArrayParameter('array_in', required=True),
        NumberParameter('factor', required=True)
    ]
```

```
    outputs = [
        NdArrayParameter('array_out')
    ]

    def execute(array_in, factor):
        return array_in * factor
```

Normally, the `execute` method will never be called directly. Instead, the `__call__` method of the base `Task` class is called; it validates and cleans the parameters (e.g., converting the string `"3"` into the number `3` for a `NumberParameter` input), then calls `execute` with the cleaned values.

Default Inputs

To specify a default value for a task input, set `required=False` on the parameter, and provide a default value for it in the `execute` method.

```
class MultiplyArray(Task):
    """ Multiply the values in an array by a given factor """

    name = 'multiply_array'

    inputs = [
        NdArrayParameter('array_in', required=True),
        NumberParameter('factor', required=False)
    ]

    outputs = [
        NdArrayParameter('array_out')
    ]

    def execute(array_in, factor=5):
        return array_in * factor
```

Multiple Return Values

If your task has multiple return values, return a `ParameterCollection` object. `ParameterCollection` behaves like a dictionary; you can set your return values like you would a dict object.

```
class Divide(Task)
    """ Perform a divide operation an return value and remainder """

    name = 'divide'

    inputs = [
        IntParameter('numerator', required=True),
        IntParameter('denominator', required=True)
    ]

    outputs = [
        IntParameter('result'),
        IntParameter('remainder')
    ]

    def execute(numerator, denominator):
```

```
        output = ParameterCollection(self.outputs)

        output['result'] = numerator // denominator # Integer division
        output['remainder'] = numerator % denominator

        return output
```

Workflows

Todo.

REST API

The REST API allows clients to run tasks and workflows as jobs, query job status as they run, and retrieve results once jobs are finished.

Registering Jobs

A job is simply a task or workflow that has been made available through the REST API. Tasks and workflows are made available as jobs with the *NC_REGISTERED_JOBS* setting. The `NC_REGISTERED_JOBS` is a dictionary of registered jobs:

```
NC_REGISTERED_JOBS = {
    'task_job': {
        'type': 'task,
        'task': 'myapp.ncdjango_tasks.SomeTask'
    },
    'workflow_job': {
        'type': 'workflow',
        'path': os.path.join(BASE_DIR, 'myapp/workflows/some_workflow.json')
    }
}
```

A basic job registration requires three things:

1. The job name (this is the dictionary key). This is the name that client will use when running the job.
2. The job type. This is either `task` or `workflow`.
3. The task, or path to the workflow file. For tasks, this is a module path. For workflows, it's the absolute path to the workflow's JSON file.

Automatically Publishing Results

Jobs can be optionally configured to publish raster results as map services by adding an extra key to the job configuration:

```
NC_REGISTERED_JOBS = {
    'some_job': {
        'type': 'task',
        'task': 'myapp.ncdjango_tasks.SomeTask',
```

```
        'publish_raster_results': True
    }
}
```

If the job returns raster results, ncdjango will automatically write the results to NetCDF datasets and publish them as temporary services. It will then return the newly created service name as the value of those output field.

By default, a black-to-white gradient will be used as the default renderer for the service. You can also specify a renderer to use for published results:

```
from clover.render.renderers.stretched import StretchedRenderer
from clover.utilities.color import Color

NC_REGISTERED_JOBS = {
    'some_job': {
        'type': 'task',
        'task': 'myapp.ncdjango_tasks.SomeTask',
        'publish_raster_results': True,
        'results_renderer': StretchedRenderer([
            (0, Color(255, 0, 0)),
            (100, Color(0, 0, 255))
        ])
    }
}
```

Note: See https://github.com/consbio/clover/tree/master/clover/render/renderers for more information on available renderers.

`results_renderer` can also be a function which returns a renderer. The function will be called with the output raster.

```
NC_REGISTERED_JOBS = {
    'some_job': {
        'type': 'task',
        'task': 'myapp.ncdjango_tasks.SomeTask',
        'publish_raster_results': True,
        'results_renderer': lambda raster: StretchedRenderer([
            (raster.min(), Color(255, 0, 0)),
            (raster.max(), Color(0, 0, 255))
        ])
    }
}
```

Cleaning up Temporary Services

To clean up temporary services. Run the celery task `ncdjango.geoprocessing.celery_tasks.cleanup_temporary_services`. You can run this directly as a function, in the background as a celery task, or set it up to run periodically using celery beat. The function will delete any temporary services older than *NC_MAX_TEMPORARY_SERVICE_AGE*.

Using the API

The API allows clients to do two things: execute jobs, and query job status, including outputs once the job has completed.

Execute a Job

To execute a job, make a `POST` request to `/geoprocessing/rest/jobs/` with two fields: the registered job name, and JSON-encoded inputs:

```
{
    "job": "some_job",
    "inputs": "{\"in\": 5}"
}
```

Note: The `inputs` field must be a string containing an encoded JSON object, rather than part of the JSON object used for the request.

Note: If you have CSRF protection enabled, you will also need to send a valid CSRF token using the `X-CSRFToken` header, or sending a `csrfmiddlewaretoken` form parameter.

The API will return information about the newly created job, including the UUID which can be used to query job status:

```
{
    "uuid": "aa346c90-68e5-4d19-a7f3-a54f6b87ec34",
    "job":"some_job",
    "created": "2016-09-02T23:36:10.768937Z",
    "status": "pending",
    "inputs": "{\"in\": 5}",
    "outputs": "{}"
}
```

Query Job Status

To query job status, make a `GET` request to `/geoprocessing/rest/jobs/<uuid>/` using the `uuid` value returned from the initial request to execute the job. The response will be identical, but the status will change as the job executes and finishes, and after it's succeeded, outputs will be provided.

```
GET /geoprocessing/rest/jobs/aa346c90-68e5-4d19-a7f3-a54f6b87ec34/
```

```
{
    "uuid": "aa346c90-68e5-4d19-a7f3-a54f6b87ec34",
    "job":"some_job",
    "created": "2016-09-02T23:36:10.768937Z",
    "status": "started",
    "inputs": "{\"in\": 5}",
    "outputs": "{}"
}
```

A jQuery Example

```
var data = {
    job: 'some_job',
    inputs: JSON.stringify({'in': 5})
};

$.post('/geoprocessing/rest/jobs/', data).success(function(data) {
    pollJobStatus(data.uuid);
});

function pollJobStatus(uuid) {
    $.get('/geoprocessing/rest/jobs/' + uuid + '/').success(function(data) {
        if (data.status === 'success') {
            var outputs = JSON.parse(data.outputs);
            // Do something with job outputs
        }
        else if (data.status === 'pending' || data.status === 'started') {
            setTimeout(function() { pollJobStatus(uuid) }, 1000);
        }
        else {
            // Handle error
        }
    });
}
```

Loading Service Data with *RasterParameter*

For tasks with a `RasterParameter` or `NdArrayParameter` input, the client can pass a reference to a published service which will be automatically loaded into memory as a `Raster` object and passed to the task as an input. To do this, the client should pass, as the input, a string with the following format: `service://<service name>:<variable name>@<timestamp (optional)>`. The timestamp is a Unix-style timestamp representing the seconds since January 1, 1970.

Note: The Unix-style timestamp is represented in seconds, unlike the Java/JavaScript timestamp, which is represented in milliseconds. Therefore timestamps from Java or JavaScript need to be divided by 1000.

In this example, a job is created where an input called `data` accepts a raster parameter, which will be filled with data from the `tmax` variable of a service called `climate-service` with the timestamp `1501895290`.

```
{
    "job": "some_job",
    "inputs": "{\"data\": \"service://climate-service:tmax@1501895290\"}"
}
```

CHAPTER 4

Reference

Models

class ncdjango.models.**Service**

A service maps to a single NetCDF dataset. Services contain general metadata (name, description), and information about the data extend, projection, and support for time.

name
The service name to be presented via web interfaces.

description
A description of the service, to be presented via web interfaces.

data_path
The path to the NetCDF dataset, relative to *NC_SERVICE_DATA_ROOT*.

projection
The data projection, as a PROJ4 string.

full_extent
A bounding box representing the full extent of the service data.

initial_extent
A bounding box representing the initial extent of the service.

supports_time
Does this service support time?

time_start
The first time step available for this service.

time_end
The last time step available for this service.

time_interval
The number of `time_interval_units` between each step.

time_interval_units

The units used for `time_interval`. Can be one of:

- `milliseconds`
- `seconds`
- `minutes`
- `hours`
- `days`
- `weeks`
- `months`
- `years`
- `decades`
- `centuries`

`calendar`

The calendar to use for time calculations. Can be one of:

- `standard` (Standard, gregorian calendar)
- `noleap` (Like the standard calendar, but without leap days)
- `360` (Consistent calendar with 30-day months, 360-day years)

`render_top_layer_only`
If `True` for multi-variable services, only the top layer will be rendered by default. Defaults to `True`.

class `ncdjango.models.`**`Variable`**
A variable in a map service. This is usually presented as a layer in a web interface. Each service may have one or more variables. Each variable maps to a variable in the NetCDF dataset.

`time_stops`
Valid time steps for this service as a list of datetime objects. **(read-only)**

`service`
Foreign key to the *`Service`* model.

`index`
Order of this variable in a list.

`variable`
Name of the variable in the NetCDF dataset.

`projection`
The data projection, as a PROJ4 string.

`x_dimension`
The name of the x dimension of this variable in the NetCDF dataset.

`y_dimension`
The name of the y dimension of this variable in the NetCDF dataset.

`name`
The variable name to be presented via web interfaces.

`description`
A description of the variable, to be presented via web interfaces.

renderer
The default renderer to use for this variable. See https://github.com/consbio/clover/tree/master/clover/render/renderers for available renderers.

full_extent
A bounding box representing the full extent of the variable data.

supports_time
Does this variable support time?

time_dimension
The name of the time dimension of this variable in the NetCDF dataset.

time_start
The first time step available for this variable.

time_end
The last time step available for this variable.

time_steps
The number of time steps available for this variable.

class `ncdjango.models.`**ProcessingJob**
An active, completed, or failed geoprocessing job.

status
The status of the celery task for this job. **(read only)**

uuid
A unique ID for this job. Usually provided to the client to query the job status.

job
The registered name of the job. See *NC_REGISTERED_JOBS*.

user
A foreign key to the `User` model, or `None` if the user is not logged in.

user_ip
The IP address of the user who initiated the job.

created
When the job was created.

celery_id
The celery task ID.

inputs
A JSON representation of the job inputs.

outputs
A JSON representation of the job outputs.

class `ncdjango.models.`**ProcessingResultService**
A result service is created from the raster output of a geoprocessing job. This model tracks which services are automatically generated from job results.

job
A foreign key to the *`ProcessingJob`* model.

service
A foreign key to the *`Service`* model.

is_temporary
Temporary services will be cleaned up when the `ncdjango.geoprocessing.celery_tasks.cleanup_temporary_services` celery task is run if they are older than *NC_MAX_TEMPORARY_SERVICE_AGE*.

created
The date the result service was created.

Settings

NC_ALLOW_BEST_FIT_TIME_INDEX

If `True` (default), find the closest valid time step to the timestamp given. If `False`, exact timestamps are required, and a timestamp which doesn't match any time step in the dataset will be considered invalid.

```
NC_ALLOW_BEST_FIT_TIME_INDEX = True
```

NC_ARCGIS_BASE_URL

The base URL for the *ArcGIS REST API* interface. Defaults to `arcgis/rest/`

```
NC_ARCGIS_BASE_URL = 'arcgis/rest/'
```

NC_ENABLE_STRIDING

Stride data if the data resolution is larger than the requested image resolution. Defaults to `False`.

```
NC_ENABLE_STRIDING = False
```

NC_FORCE_WEBP

Return `WebP`-formatted images instead of PNG if the browser supports it, regardless of requested format. Defaults to `False`.

```
NC_FORCE_WEBP = False
```

NC_INSTALLED_INTERFACES

A list of web services interfaces to enable. By default, this is the *ArcGIS REST API* (plus the *extended ArcGIS API*) and the *data* interface.

```
NC_INSTALLED_INTERFACES = (
    'ncdjango.interfaces.data',
    'ncdjango.interfaces.arcgis_extended',
    'ncdjango.interfaces.arcgis'
)
```

NC_MAX_TEMPORARY_SERVICE_AGE

The length of time (in seconds) to keep a temporary service (usually created as the result of a geoprocessing job) before automatically deleting it. Defaults to `43200` seconds (12 hours).

```
NC_MAX_TEMPORARY_SERVICE_AGE = 43200  # 12 hours
```

NC_MAX_UNIQUE_VALUES

The maximum number of unique values for a dataset to return through the *data* interface. Defaults to `100`.

```
NC_MAX_UNIQUE_VALUES = 100
```

NC_REGISTERED_JOBS

A list of geoprocessing jobs to make available to clients. This should be a dictionary with the following format:

```
NC_REGISTERED_JOBS = {
    '<name>': {  # Name used for the API
        'type': '<task|workflow>',  # Job type: 'task' or 'workflow'
        'task': '<module path to task class>',  # If type is task
        'path': '<absolute path to workflow definition file>',  # If type is workflow
        'publish_raster_results': True,  # Automatically publish raster outputs as␣
↪services?
        'results_renderer': StretchedRenderer([
            (0, Color(240, 59, 32)),
            (50, Color(254, 178, 76)),
            (100, Color(255, 237, 160))
        ])  # Renderer definition for automatically published services
    }
}
```

NC_SERVICE_DATA_ROOT

The root location of NetCDF datasets. Defaults to `/var/ncdjango/services/`.

```
NC_SERVICE_DATA_ROOT = '/var/ncdjango/services/'
```

NC_TEMPORARY_FILE_LOCATION

The location to store temporary files (uploads). Defaults to `/tmp`.

```
NC_TEMPORARY_FILE_LOCATION = '/tmp'
```

NC_WARP_MAX_DEPTH

The maximum recursion depth to use when generating the mesh used to warp output images to the requested projection. Defaults to `5`.

```
NC_WARP_MAX_DEPTH = 5
```

NC_WARP_PROJECTION_THRESHOLD

The tolerance (in pixels) to use when warping images to the requested projection. Defaults to `1.5`. When warping the image, a mesh of varying size is used. The size is determined by recursively subdividing a line and comparing the projected midpoint to a "guessed" midpoint. The subdivision stops when the difference is within the tolerance, or `:ref:`setting-warp-max-depth`` is reached.

```
NC_WARP_PROJECTION_THRESHOLD = 1.5
```

Python Module Index

n

Index

Y